The Prize-collecting Steiner Forest Problem

Kiss Bendegúz
Supervisor: Dr. Király Tamás

1 Introduction

The Prize-collecting Steiner Forest Problem (often written as PCSF) is a less known problem in graph theory. The goal here is to find a forest in an undirected graph, which minimizes the following costs: we pay for the edges included in our forest and we pay for those node pairs, which are not connected in our solution. There can be different descriptions of the problem, for instance, the penalties are given on the nodes and not on the node pairs, but we will focus on the case mentioned earlier. We will give a proper definition later. Considering the problem, it seems logical to examine it mainly from a theoretical point of view, but we will show some real life application of PCSF in analyzing biological networks, which we found quite interesting.

This semester I have completed the following tasks. Firstly, I started working on a heuristic approach that was used in biological networks. I kept the main ideas of the algorithm, but made some changes in a few steps because the PCSF problem defined in the paper is a little bit different from ours. Secondly, I improved the running time of the 2-approximation algorithm I have studied so far. Finally, I tested my implementations on different random graph models with randomly generated edge costs and penalties. The results will be shown in the last section.

2 Description of PCSF

Definition: 1 *Given an undirected $G = (V, E)$ with non-negative edge costs $c : E \mapsto \mathbb{R}_0^+$ and a non-negative penalty function $\pi : V \times V \mapsto \mathbb{R}_0^+$ that is 0 over node pairs (i, j), where $i \geq j$. The goal is to find a forest minimizing the following expression:*

$$\sum_{e \in F} c_e + \sum_{(i,j) \in Q} \pi_{ij}$$

where Q contains all disconnected node pairs in F.

A simpler version of this problem is the Steiner Forest Problem, where we are given specific node pairs called demands that we must satisfy by connecting the two nodes in each demand. It is generalized from the well-known Steiner Tree problem, where we search for a tree including specified nodes called terminals. The generalization can be made by having a demand for each terminal pair.

3 Algorithms for PCSF

In this section, I briefly explain the algorithms I studied during the three semesters. There will be an approximation algorithm with an approximation ratio of 3 from which we can easily obtain 2 by running the same algorithm twice with some modifications and comparing the two solutions. After that, we take a look at a fast heuristic approach, which was mainly made for optimizing biological networks.

3.1 The 3-approximation algorithm

Since the problem is NP-hard, we cannot give an exact solution in polynomial time, but we can give a solution which does not cost more than twice as much as the optimal solution by the following approximation algorithm. It is more like a theoretical result. In a practical view, it is not really efficient because of its slow running time on larger inputs.

The procedure is building a solution by coloring edges using a predetermined schema. The fully-colored edges will be added to our final solution. The coloring procedures are categorized into two types: static and dynamic coloring. The algorithm maintains a family of active sets, denoted as $ActS$, which initially contains the nodes as sets with one element. These sets will be the source of our coloring process.

During static coloring, we decide which is the most time we can color the outgoing edges of the active sets, so we do not 'overcolor' any of the edges (The coloring time of an edge must not exceed its cost). If an edge connects two active sets, it will be colored twice as fast as the others. We store this duration as Δ_e in each iteration.

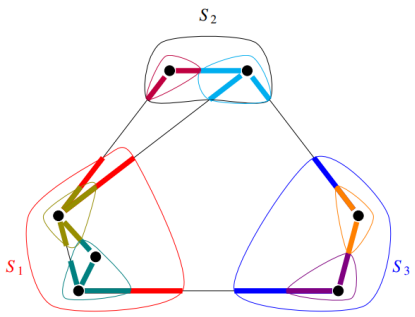

Figure 1: The visualization of a static coloring process

The much more difficult coloring schema is called dynamic coloring which is not that easy to visualize as in the static case. In this procedure, we take the penalties into account. Since the penalties are given on node pairs, the previous method does not work, so we want to mimic a static coloring with penalty constraints. We construct a graph called SetPairGraph to compute the proper duration. There are nodes representing every set that colored its outgoing edges during the algorithm. We also have nodes for each node pair. Finally, we add a sink and a source node. The edge set contains edges from the source to every node that represents a set with capacity y_S where y_S means how long the S set colored its outgoing edges, edges from nodes representing the (i, j) node pairs to the sink with capacity π_{ij} and edges with infinite capacity between a node represented by a set and a node represented by a pair if the set cuts the node pair. Coloring with an S set can be demonstrated in this graph by increasing the edge capacity from the source to the node representing S and running a max-flow algorithm. We do this for every active set.

If each edge which enters a node representing an active set becomes tight, we call this coloring valid. We search for the largest value (coloring duration), which we can increase the edge capacities with obtaining a valid coloring. This duration is stored as Δ_p.

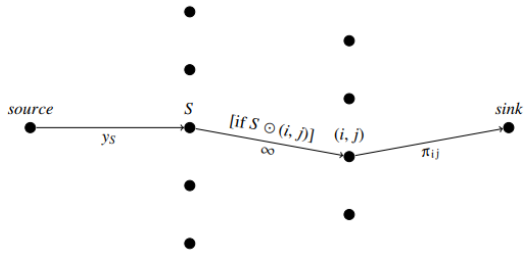

Figure 2: The SetPairGraph

After computing Δ_e and Δ_p, we define $\Delta := \min\{\Delta_e, \Delta_p\}$. We color the outgoing edges of the active sets for Δ time, and the fully-colored edges will be added to our forest. We update $ActS$ as follows. If a fully-colored edge connects two active sets, we delete them and add the union to $ActS$. If it connects an active set and a deactivated set (defined later), we delete the active set and add the union of these two sets to $ActS$.

It is possible that sets in $ActS$ cannot color their outgoing edges anymore after static or dynamic coloring occurs. These sets will be deactivated (removed from $ActS$) before the next iteration. To decide whether an S set needs to be deactivated, we use our SetPairGraph mentioned above. If we can increase the capacity of the edge entering the node representing S such that the max-flow value increases, this means that the S set can color in the next iteration. Otherwise, the set becomes tight. We continue this coloring procedure until no active sets remain. We mention that the whole node set cannot be considered an active set because it does not have any outgoing edges.

At the end of the algorithm, we have two remaining steps. Firstly, we have to select the node pairs for which we are willing to pay their penalties. These pairs are called tight pairs. There is a procedure, which tries to decrease the number of tight pairs with the help of non-tight pairs and our SetPairGraph. We pay penalties for the remaining tight pairs. Secondly, after we decided which pairs must be connected, we try deleting edges from our solution by maintaining the connection between the non-tight pairs. The remaining forest will be our solution.

The running time of the algorithm is polynomial, but strongly relies on the graph structure and the running time of the subroutines we use, for instance, for computing max-flow. This will be explained in the implementation section.

3.2 The 2-approximation algorithm

We can improve the approximation ratio of this algorithm to 2 by running it again with some modifications. If we paid a penalty for a pair in the previous run we set the penalty 0 for these pairs. After this slight modification we run the 3-approximation algorithm again and evaluate the solution with the original penalties. We compare the two solutions and return the better one. It can be shown that this will be a solution with cost, not larger than twice the optimal cost.

3.3 Fast-heuristic for PCSF

In the previous semester, I worked on a heuristic approach, but it only worked well on specific inputs and gave really bad solutions on other inputs. Fortunately, I came across a much better heuristic approach, which was not exactly made for our problem, but the main ideas were enough to develop a heuristic algorithm. The approach described in the paper was originally made for biological applications, which I will describe briefly in the next section.

The main idea of the algorithm is to categorize the nodes into clusters considering the ratio of the cost of the shortest path and the penalty between node pairs, and find a minimum spanning tree in each cluster. The algorithm has two phases: a clustering phase and an MST phase.

First of all, we need to prepare some data. We use a D matrix, in which $D_{i,j}$ is the cost of the shortest path between i and j in G. It can be easily computed by the Floyd-Warshall algorithm in $O(n^3)$ time. We need to collect the nodes, which have at least one non-negative penalty with another node. These nodes will be the terminal nodes stored in a set U. We will use U in the clustering phase. The other nodes are called Steiner nodes, which will be useful in the MST phase. In the original algorithm, the penalties are on the nodes and the nodes with non-negative penalty will be the terminal nodes.

We can start the clustering phase after these preparations. This part of the algorithm starts with the set U representing the unassigned nodes. We take a random element $u \in U$ and assign it to a new cluster C_u. Now, we decide whether a $v \neq u$, $v \in U$ node can be assigned to this cluster. To decide, we check the following inequality:

$$\alpha \cdot D_{u,v} \leq \pi_{u.v}$$

If a $v \in U$ node satisfies this inequality, then we assign it to the cluster, C_u and delete v from U because it became assigned. The α parameter is given at the beginning of the algorithm. To get reasonable solutions we choose this parameter from the $(0, 1]$ interval. In the testing section we will see which α values gave the better solutions. We keep making clusters until no unassigned nodes remain.

We apply an additional step, which was not described in the paper. We try to contract some clusters to get better solutions. If we want to contract two clusters, we check the following inequality.

$$c(MST(C_1)) + c(MST(C_2)) \geq c(MST(C_1 \cup C_2)) \quad \text{for clusters} \quad C_1, C_2$$

where $c(MST(U))$ is the cost of the minimum spanning tree in $G[U]$. To make these contractions fast we make a graph where each node represents a cluster and we add edges between clusters if they satisfy the previous inequality together. The components of this graph will be contracted into new clusters. After this we can continue with the next phase of the algorithm.

The second phase is the MST phase, where we use our clustering given by the previous phase. We start with some initialization. For each subgraph $G[C_v]$ denoted as G_v, we construct its metric closure, which means a complete graph on the nodes of G_v and the cost of an uv edge is $D_{u,v}$. We take the union of these graphs and add a root node which we connect to every other node. The weight of these edges is another parameter denoted by ω. The number of components in the final solution will depend on this parameter. More details about this will be in the testing section. In this new graph, we search for a minimum spanning tree T.

We are ready to obtain our solution. We start with an empty graph with the nodes of the original graph (including the Steiner-nodes). An uv edge in T represents an uv path in G. We take all these paths represented by the edges of T and add all their edges to our empty graph. Finally, we search for a minimum spanning forest in this graph, and this will be our output.

4 Applications in biological networks

In this section, I present a very interesting application of the problem. Biological networks in a cell can be represented as graphs. We call this graphs, interactomes.The nodes of these graphs are biomolecules, such as proteins. The edges represent all physical interaction between the molecules. We can assign weights to these edges as well. These weights can be assigned from the time we spend on examining an interaction, or from the cost of the equipment we use for examination, but there may be other reasonable weight choices.

Let us look at the following example. We want to examine a disease by checking the interactions of some proteins in a cell. Some protein interactions contain useful information about the disease, some of them are not. We can define a PCSF problem on the interactome of the biological network in the cell. The edge weights are given similarly as we discussed. The value of the penalties will be decided by the importance of a protein-protein interaction. The less important pairs get low penalties, the more important ones get big penalties. We can imagine the penalties as the information loss, if we do not check the interaction between the proteins represented by the node pair. Now the problem is to find a PCSF solution in the interactome with the given edge weights and penalties. This actually means that we want to get the most information about the disease by checking as much as possible interactions, but we do not want to pay much for the examination. The heuristic algorithm I studied, was primarily made for these kind of biological applications.

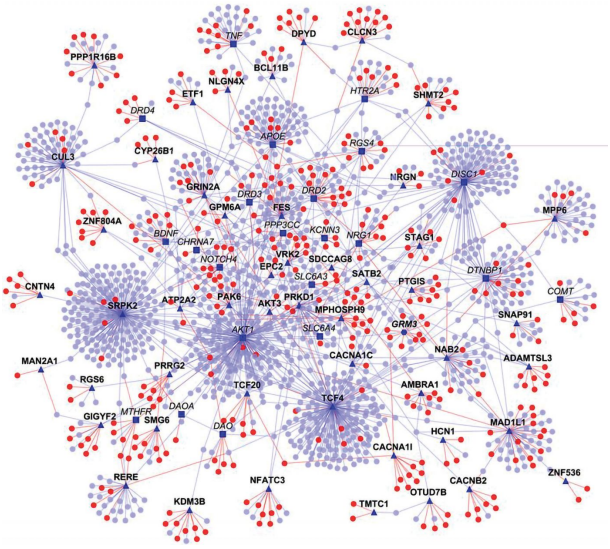

Figure 3: A protein interactome

5 Implementation

After I had understood the main ideas of the algorithms, I started to implement them in Python using the `networkx` library for storing graphs and its built-in graph algorithms to compute max-flow or to find a minimum spanning tree for instance.

5.1 Approximation algorithms

I finished implementing the approximation algorithm in the previous semester, but It needed some improvements especially in the running time, so I had to reconsider some step of the program. The main function called `pcsf3`, uses several subroutines. In `find_delta_e` we calculate the Δ_e value mentioned in the algorithm. This part always fast because we only iterate through the edges. The `find_delta_p` function does the similar task as the previous subroutine, but for Δ_p. It is a much more complex computation because it needs that specific graph, SetPairGraph mentioned before, and also to compute a max-flow (for this I used a built-in function). The graph is constructed by the `set_pair_graph` function once and during the algorithm we just modify it if needed. For example, adding new nodes representing the new active sets and changing the capacities during the computation of Δ_p or deciding tightness for a set. In the previous version I constructed the graph in each iteration, and it took a long time sometimes, so the change helped a lot considering the runtime. There is a subroutine called `check_set_is_tight`, which decides the tightness of a set during the algorithm. It is a bit slow because it has to run a max-flow algorithm for each active set. The `reduce_tight_pairs` function collects all pairs, we must pay penalties for. The runtime of this subroutine is not that slow because we only check $O(n^2)$ tight pairs. The main issue was storing the coloring durations. So far, I used a tuple list denoted by y storing the data of all coloring that occurs during the algorithm. The tuples contained the set that colors, the edge colored by the set and the duration. This data structure seemed usable, but I found out that some computations take a long time using this list. I changed y for a DataFrame, where the columns represent the edges and the rows represent the sets that color during the algorithm and $y_{S,e}$ is the duration of the coloring by the S set on the e edge. It was much more useful. For example, we can easily check whether an edge is colored, by adding each element in its column. I also stored the coloring durations of the sets in a dictionary. These data structures usually do not have huge size because there are only $O(n)$ sets that can become active during the algorithm. The `pcsf2` function just uses the `pcsf3` function with the modifications mentioned before, to generate our second solution.

5.2 Heuristic approach

This semester, I mainly worked on the implementation of the heuristic approach. There is not much to say about the code because it just a simple implementation of the algorithm without the use of special data structures. The code has a main function `fast_heuristic_pcsf`. It uses one subroutine the `clustering_phase` function, which makes the clusters for the MST-phase of the algorithm. It also contains the additional contracting step as a subroutine. The D matrix is computed by the built-in `networkx` Floyd-Warshall algorithm. During the procedure we need to find a lot of MST-s. For that I used a built-in `networkx` function as well.

6 Testing

6.1 Testing instances

I tested the algorithms on small graphs with 20 nodes and larger graphs with 100 nodes using different random graph models. I generated the edge costs uniformly from a given interval, which was [1,10] for small instances ($n = 20$) and [1,20] for large instances ($n = 100$). Larger intervals generate more edges with different cost, so that the components are built much slower in `pcsf3`, causing huge running times. The penalties were given with probability of $\frac{1}{5}$ for 20 nodes and with probability of $\frac{1}{10}$ for 100 nodes, and the value was 1. If we want interesting solutions, we do not want to set the penalty value high.

I tested the heuristic approach using the $\alpha = 1/i \quad i = 1 \dots 10$ values and chose the best solution. The parameter ω was 1 for all instances.

The first model I used was the Erdős-Rényi model. I set the probability parameter $\frac{1}{2}$ for small instances and $\frac{1}{10}$ for large instances.

The second model I used was the Barabás-Albert model with $m = 2$, where the parameter m is the number of old nodes we connect with the additional nodes.

6.2 Results

I summarize the results in two tables, one for Erdős-Rényi instances and one for Barabás-Albert instances. Compared to the previous semester, the running times for Erdős–Rényi instances improved significantly. The Barabás-Albert graphs are usually sparse, so the approximation algorithms build components much slower, which causes longer running time, but they give more interesting solutions in these instances. The heuristic approach solutions are usually better than the approximation algorithm solutions in the smaller instances, but on 100 nodes the heuristic solutions tend to be much worse in some cases.

n	PCSF3			PCSF2			Heuristic		
	Cost	Penalty	Time (s)	Cost	Penalty	Time (s)	Cost	Penalty	Time (s)
20	56	4	6.701	56	4	9.499	0	44	0.412
20	24	16	3.107	8	26	2.430	0	33	0.188
20	0	40	2.465	0	40	0.484	0	40	0.121
20	35	6	2.313	35	6	1.687	23	15	0.035
20	59	6	4.085	38	23	6.715	0	43	0.198
20	42	4	3.582	42	4	4.645	36	9	0.028
100	394	31	684.959	394	31	1339.677	158	302	3.391
100	463	11	828.725	463	11	1339.677	0	480	7.816
100	451	12	830.526	451	12	1200.942	0	472	8.672
100	432	17	1008.810	0	487	337.887	0	487	6.899
100	455	12	899.101	455	12	1193.687	0	508	11.937
100	431	3	856.879	431	3	1252.067	162	330	3.973

Table 1: Results on Barabási-Albert random graphs

n	PCSF3			PCSF2			Heuristic		
	Cost	Penalty	Time (s)	Cost	Penalty	Time (s)	Cost	Penalty	Time (s)
20	31	6	3.507	7	22	1.534	0	26	0.121
20	30	12	3.460	22	17	2.767	0	36	0.165
20	26	2	1.621	26	2	1.280	30	0	0.023
20	31	0	2.755	31	0	1.718	31	0	0.041
20	25	0	1.102	25	0	0.598	25	0	0.043
20	37	0	1.918	37	0	1.803	23	12	0.049
100	322	0	468.008	322	0	465.741	279	95	2.002
100	276	0	391.126	276	0	412.005	231	60	3.158
100	301	0	466.018	301	0	454.419	265	73	1.797
100	302	0	525.471	302	0	588.565	283	22	1.524
100	293	5	540.143	293	5	799.911	226	105	2.361
100	309	0	603.394	309	0	612.354	266	50	2.603

Table 2: Results on Erdős-Rényi random graphs

References

[1] Ahmadi, Ali, et al. "2-approximation for prize-collecting Steiner forest." Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). Society for Industrial and Applied Mathematics, 2024.

[2] Akhmedov, Murodzhon, et al. "A fast prize-collecting steiner forest algorithm for functional analyses in biological networks." International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. Cham: Springer International Publishing, 2017.